1. Docker :
   1. Container vs virtual machine
   2. Docker engine
   3. Docker Objects
   4. Docker underlying technogies
2. Docker commands
   1. Launch simple application
   2. Docker commands
   3. Private and public registry
   4. Dockerfile
   5. Docker compose
3. Networking
   1. Host
   2. Bridge
   3. None
   4. Iptables
4. Swarm
   1. Understand and Setup the Swarm Cluster
   2. Deploy sample app

**Docker:**

It is a tool that packages up an application and all its dependencies in a “virtual container” so that it can be run on any Linux system or distribution.

**When would I use Docker?**

* Configuration Simplification
* Enhance Developer Productvity
* Server Consolidation and Management
* Application Isolation
* Rapid Deployment
* Build Management

**Does docker run on Linux only?**

### **Containers VS Virtual machines:**

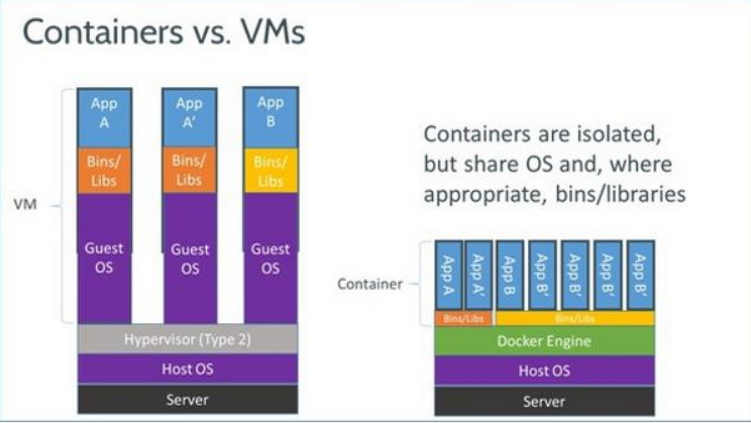
a virtual machine (VM), is an emulation of a specific computer system.Virtualization software allows you to set up one operating system within another. Although they both share the same physical hardware. the virtual machine is isolated from that hardware and has to communicate with it through something called a Hypervisor.

By contrast, a virtual machine (VM) runs a full-blown “guest” operating system with virtual access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.

Container:

It is an entirely isolated set of packages, libraries and/orapplications that are completely independent from its surroundings.A container runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

Cost: How many vm/containers can launch on single host?



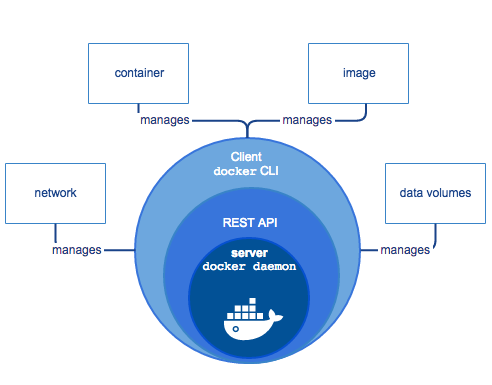
Containerization is increasingly popular because containers are:

* Flexible: Even the most complex applications can be containerized.
* Lightweight: Containers leverage and share the host kernel.
* Interchangeable: You can deploy updates and upgrades on-the-fly.
* Portable: You can build locally, deploy to the cloud, and run anywhere.
* Scalable: You can increase and automatically distribute container replicas.
* Stackable: You can stack services vertically and on-the-fly.

## **Docker Engine = Architecture**

Docker Engine is a client-server application with these major components:

* A server which is a type of long-running program called a daemon process (the dockerd command).
* A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
* A command line interface (CLI) client (the docker command).



The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI.

The daemon creates and manages Docker objects, such as images, containers, networks, and volumes.

### **The Docker daemon**

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

### **Docker objects**

#### **IMAGES**

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.

#### **CONTAINERS**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

### **Docker registries**

A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

**The underlying technology**

Docker is written in [Go](https://golang.org/) and takes advantage of several features of the Linux kernel to deliver its functionality.

**Isolation: Namespaces**

Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of **isolation**. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Engine uses namespaces such as the following on Linux:

* The **pid** namespace: Process isolation (PID: Process ID).
* The **net** namespace: Managing network interfaces (NET: Networking).
* The **ipc** namespace: Managing access to IPC resources (IPC: InterProcess Communication).
* The **mnt** namespace: Managing filesystem mount points (MNT: Mount).
* The **uts** namespace:Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

**Control groups**

Docker Engine on Linux also relies on another technology called control groups (cgroups). **A cgroup limits an application to a specific set of resources**. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. For example, you can limit the memory available to a specific container.

**Union file systems: AUFS(advanced multi-layered unification filesystem)**

**Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight** and fast. Docker Engine uses UnionFS to provide the building blocks for containers. Docker Engine can use multiple UnionFS variants, including AUFS, btrfs, vfs, and DeviceMapper.

AUFS implements a union mount for Linux file systems.

**Container format**

Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a container format. The default container format is libcontainer. In the future, Docker may support other container formats by integrating with technologies such as BSD Jails or Solaris Zones.

**Docker Commands:**

Launch simple application

First Container Launch:

$ docker run -i -t ubuntu /bin/bash

Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.

Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine’s network connection.

Docker starts the container and exe cutes /bin/bash. Because the container is run interactively and attached to your terminal (due to the -i and -t) flags, you can provide input using your keyboard and output is logged to your terminal.

## **-i, --interactive Keep STDIN open even if not attached**

**-t, --tty Allocate a pseudo-TTY**

**-d, --detach Run container in background and print container ID**

Commands:

1. Launch a container name c1 , running on backgroud with image ubuntu:16.04

docker run -it -d --name c1 --restart=always -m=500m ubuntu:latest

2. Run a command on running container or execute the bash command

docker exec -it c1 bash

3. add host path volume and publish container port 80 on host 8888 port

docker run -it -d --name c2 --restart=always -v /root/website:/var/www/html -p 8888:80 basivireddy/apache2:latest

Here is a list of the basic Docker commands from this page, and some related ones if you’d like to explore a bit before moving on.

docker container ls # List all running containers

docker container ls -a # List all containers, even those not running

docker run friendlyhello # Run "friendlyname" mapping port 4000 to 80

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker pull username/repository:tag # Upload tagged image to registry

**Volumes: https://docs.docker.com/storage/volumes/**

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.

In addition, volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.

### **Remove anonymous volumes**

To automatically remove anonymous volumes, use the –rm option. For example, this command creates an anonymous /foovolume. When the container is removed, the Docker Engine removes the /foo volume but not the awesome volume.

$ docker run --rm -v /foo -v awesome:/bar busybox top

### **Remove all volumes**

To remove all unused volumes and free up space:

$ docker volume prune

**Docker Registry:**

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

**Public Registry: Docker Hub**

1. create a account in <https://hub.docker.com/>

2. Create a repository using github repository

1. Link github account to docker hub account

goto profile settings --> click Linked account and Services and link github

1. Create repository

click create button --> Create Automated build --> click on github and select github repository

1. Create a build

goto dashboard --> goto repository --> click on Build settings --> select correct branch and Trigger the build

3. Create a repository using docker CLI:

1. push local image to docker hub registry

docker push username/repository:tag

**Private Registry: untrusted registry**

docker run -d -it --restart=always -e REGISTRY\_STORAGE\_FILESYSTEM\_ROOTDIRECTORY='/data' --net=host -v /root/data:/data --name privateregistry -p 5000:5000 registry:2

docker run -d -it --name registry-web -e REGISTRY\_URL=http://18.216.4.118:5000/v2 -e REGISTRY\_NAME=18.216.4.118:5000 -p 8080:8080 hyper/docker-registry-web

echo "18.216.4.118 privateregistry" >> /etc/hosts

curl -X GET http://18.216.4.118:5000/v2/\_catalog

curl -X GET http://18.216.4.118:5000/v2/image/tags/list

echo "{ \"insecure-registries\":[\"privateregistry:5000\"] }" > /etc/docker/daemon.json

service docker restart

docker tag image privateregistry:5000/repository:tag

docker push privateregistry:5000/repository:tag

docker pull ubuntu:14.04

docker tag ubuntu:14.04 privateregistry:5000/ubuntu:14.04

docker push privateregistry:5000/ubuntu:14.04

Docker private registry

/etc/docker/daemon.json

{ "insecure-registries":["privateregistry:5000"] }

# **Tag the image:**

docker tag image privateregistry:5000/repository:tag

### **Publish the image to registry**

Upload your tagged image to the repository:

docker push privateregistry:5000/repository:tag

## **Dockerfile**

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

The docker build command builds an image from a Dockerfile and a context.

The build’s context is the set of files at a specified location PATH or URL. The PATH is a directory on your local filesystem. The URL is a Git repository location.

$ docker build [-f /path/to/a/Dockerfile] [-t image:tag] [context]

Examples:

1. To tag the image into multiple repositories after the build, add multiple -t parameters when you run

$ docker build -t shykes/myapp:1.0.2 -t shykes/myapp:latest .

If you didn’t specify the Dockerfile location. Docker search for Dockerfile in Context. Traditionally Dockerfile located in the root of the context.

Dockerfile Format

INSTRUCTION arguments

* The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Dockerfile Commands:

1. FROM

A Dockerfile must start with a `FROM` instruction. The FROM instruction specifies the Base Image from which you are building.

1. Comment lines start with #
2. COPY

copy the file from host to container

1. ADD :

ADD has two forms:

ADD [--chown=<user>:<group>] <src>... <dest>

ADD [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

Note: The --chown feature is only supported on Dockerfiles used to build Linux containers, and will not work on Windows containers. The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

1. WORKDIR:

Set the working directory

1. VOLUME

VOLUME ["/data"]

The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

1. RUN:

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile

RUN has 2 forms:

* RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux)
* RUN ["executable", "param1", "param2"] (exec form)

1. LABEL

LABEL <key>=<value> <key>=<value> <key>=<value> ...

The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.

A few usage examples:

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL version="1.0"

LABEL description="This text illustrates \

that label-values can span multiple lines."

1. EXPOSE

EXPOSE <port> [<port>/<protocol>...]

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the -p flag on docker run to publish and map one or more ports, or the -P flag to publish all exposed ports and map them to high-order ports.

By default, EXPOSE assumes TCP. You can also specify UDP:

EXPOSE 80/udp

1. ENV

ENV <key> <value>

ENV <key>=<value> ...

The ENV instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage

11. CMD:

The CMD instruction has three forms:

* CMD ["executable","param1","param2"] (exec form, this is the preferred form)
* CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
* CMD command param1 param2 (shell form)

There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect.

The main purpose of a CMD is to provide defaults for an executing container

12. ENTRYPOINT

ENTRYPOINT has two forms:

* ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)
* ENTRYPOINT command param1 param2 (shell form)

An ENTRYPOINT allows you to configure a container that will run as an executable.

Example:

1.

FROM debian:stable

RUN apt-get update && apt-get install -y --force-yes apache2

EXPOSE 80 443

VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

## **Build the app**

Now run the build command. This creates a Docker image, which we’re going to tag using -t so it has a friendly name.

docker build -t apache .

Docker compose:

The Compose file is a YAML file which defines services, networks, and volumes.

<https://docs.docker.com/compose/gettingstarted/>

**Services**:

In a distributed application, different pieces of the app are called “services.” For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on.

Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Compose is a tool for defining and running multi-container Docker applications.

Create a file called docker-compose.yml in your project directory and paste the following:

version: '2'

services:

web:

image: gitea/gitea:1.3.2

volumes:

- ./data:/data

ports:

- "3000:3000"

- "2222:22"

depends\_on:

- db

restart: always

db:

image: mariadb:10

restart: always

environment:

- MYSQL\_ROOT\_PASSWORD=changeme

- MYSQL\_DATABASE=gitea

- MYSQL\_USER=gitea

- MYSQL\_PASSWORD=changeme

volumes:

- ./db/:/var/lib/mysql

docker-compose up

If you started Compose with docker-compose up -d, stop your services once you’ve finished with them:

$ docker-compose stop

You can bring everything down, removing the containers entirely, with the down command. Pass --volumes to also remove the data volume used by the Redis container:

$ docker-compose down --volumes

Commands:

1.verison: compose version

2.services: in bellow we can mention the services

3.image: docker images to launch application

4.volumes:

volumes:

- /root/data:/data

5. ports:

ports:

- "3000:3000"

6.restart: always

7.depends\_on :on which services  
 depends\_on:

- db

8.environment

environment:

- MYSQL\_ROOT\_PASSWORD=changeme

9.create docker image: build: .

10.command : run a command at time of running container

command: python3 manage.py runserver 0.0.0.0:8000

Examples:

1. <https://docs.docker.com/compose/gettingstarted/>
2. <http://containertutorials.com/docker-compose/index.html>

**Networking:**

Docker uses **host-private networking**. It creates a virtual bridge, called **docker0** by default, and allocates a subnet from one of the private address blocks defined in [RFC1918](https://tools.ietf.org/html/rfc1918) for that bridge. For each container that Docker creates, it allocates a virtual ethernet device (called **veth**) which is attached to the bridge. The veth is mapped to appear as **eth0** in the container, using Linux namespaces. The in-container **eth0** interface is given an IP address from the bridge’s address range.

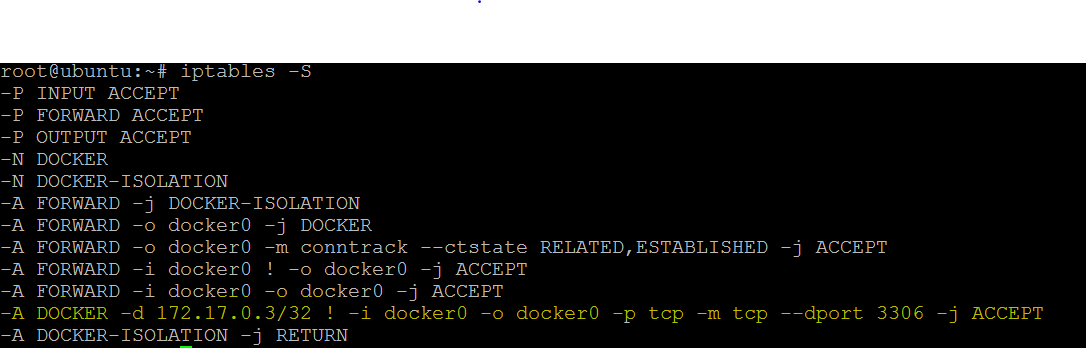
The result is that Docker containers can talk to other containers only if they are on the same machine (and thus the same virtual bridge). Containers on different machines can not reach each other - in fact they may end up with the exact same network ranges and IP addresses.

In order for Docker containers to communicate across nodes, they must be allocated ports on the machine’s own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or else be allocated ports dynamically

root@ubuntu:~# brctl show

bridge name bridge id STP enabled interfaces

docker0 8000.0242ec1cc7e4 no



## **Network drivers**

Docker’s networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

* bridge: The default network driver. If you don’t specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.** See [bridge networks](https://docs.docker.com/network/bridge/).
* host: For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly. host is only available for swarm services on Docker 17.06 and higher. See [use the host network](https://docs.docker.com/network/host/).
* overlay: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See [overlay networks](https://docs.docker.com/network/overlay/).
* macvlan: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host’s network stack. See [Macvlan networks](https://docs.docker.com/network/macvlan/).
* none: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See [disable container networking](https://docs.docker.com/network/none/).
* [Network plugins](https://docs.docker.com/engine/extend/plugins_services/): You can install and use third-party network plugins with Docker. These plugins are available from [Docker Store](https://store.docker.com/search?category=network&q=&type=plugin) or from third-party vendors. See the vendor’s documentation for installing and using a given network plugin.

# **bridge networks**

# **Differences between user-defined bridges and the default bridge**

1. **User-defined bridges provide better isolation and interoperability between containerized applications**.
2. **User-defined bridges provide automatic DNS resolution between containers**.

Containers on the default bridge network can only access each other by IP addresses, unless you use the [--link option](https://docs.docker.com/network/links/), which is considered legacy. On a user-defined bridge network, containers can resolve each other by name or alias.

1. **Containers can be attached and detached from user-defined networks on the fly**.

During a container’s lifetime, you can connect or disconnect it from user-defined networks on the fly. To remove a container from the default bridge network, you need to stop the contain

1. **Linked containers on the default bridge network share environment variables**.Originally, the only way to share environment variables between two containers was to link them using the [--link flag](https://docs.docker.com/network/links/). This type of variable sharing is not possible with user-defined networks. However, there are superior ways to share environment variables. A few ideas:
   1. Multiple containers can mount a file or directory containing the shared information, using a Docker volume.
   2. Multiple containers can be started together using docker-compose and the compose file can define the shared variables.
   3. You can use swarm services instead of standalone containers, and take advantage of shared [secrets](https://docs.docker.com/engine/swarm/secrets/) and [configs](https://docs.docker.com/engine/swarm/configs/).

Examples:

docker network create --subnet=172.16.1.0/16 --gateway=172.16.1.10 --ip-range=172.16.1.128/25 --opt com.docker.network.bridge.name=br0 my-net

$ docker create --name my-nginx --network my-net --publish 8080:80 nginx:latest

To connect a **running** container to an existing user-defined bridge, use the docker network connect command. The following command connects an already-running my-nginx container to an already-existing my-net network:

$ docker network connect my-net my-nginx

## **Disconnect a container from a user-defined bridge**

To disconnect a running container from a user-defined bridge, use the docker network disconnect command. The following command disconnects the my-nginx container from the my-net network.

$ docker network disconnect my-net my-nginx

# **Use overlay networks**

The overlay network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

When you initialize a swarm or join a Docker host to an existing swarm, two new networks are created on that Docker host:

* an overlay network called ingress, which handles control and data traffic related to swarm services. When you create a swarm service and do not connect it to a user-defined overlay network, it connects to the ingress network by default.
* a bridge network called docker\_gwbridge, which connects the individual Docker daemon to the other daemons participating in the swarm.

**Swarm:**

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single virtual host.

Docker Swarm is a clustering and scheduling tool for Docker containers. With Swarm, IT administrators and developers can establish and manage a cluster of Docker host systems, or nodes, as a single virtual system

## **Feature highlights**

Cluster management integrated with Docker Engine: Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don’t need additional orchestration software to create or manage a swarm.

Decentralized design: Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.

Declarative service model: Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application comprised of a web front end service with message queueing services and a database backend.

Scaling: For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

Desired state reconciliation: The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.

Multi-host networking: You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.

Service discovery: Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.

Load balancing: You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.

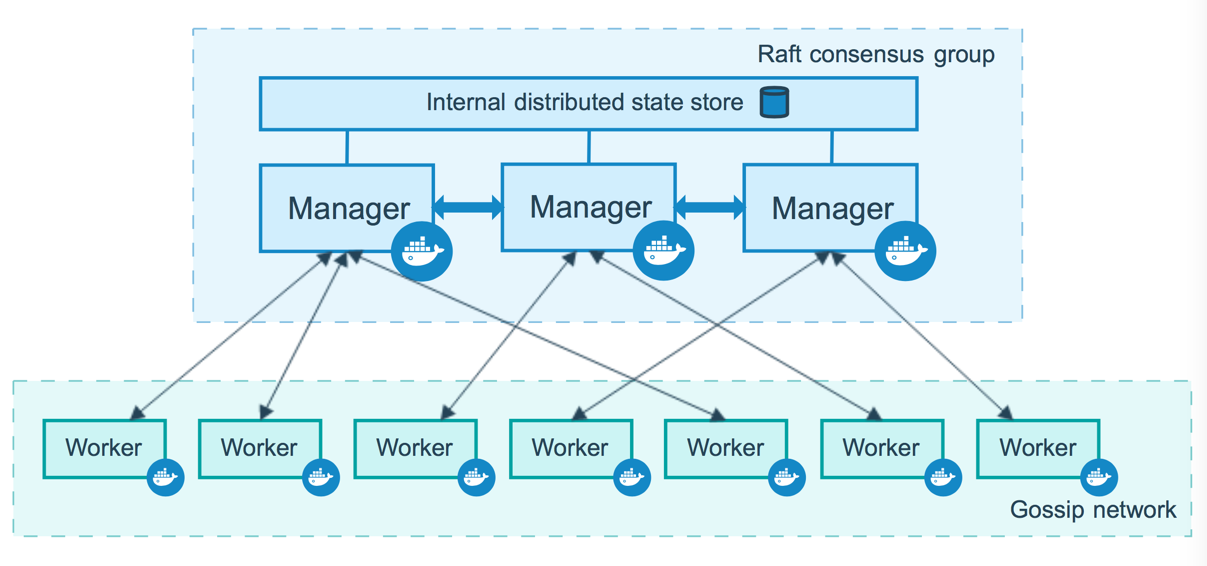
Secure by default: Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.

Rolling updates: At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll-back a task to a previous version of the service.

The cluster management and orchestration features embedded in the Docker Engine are built using [swarmkit](https://github.com/docker/swarmkit/). Swarmkit is a separate project which implements Docker’s orchestration layer and is used directly within Docker.

A swarm consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run [swarm services](https://docs.docker.com/engine/swarm/key-concepts/" \l "services-and-tasks)). A given Docker host can be a manager, a worker, or perform both roles.

Architecture:



## **Nodes**

A node is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node.

To deploy your application to a swarm, you submit a service definition to a manager node. The manager node dispatches units of work called [tasks](https://docs.docker.com/engine/swarm/key-concepts/" \l "services-and-tasks) to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default manager nodes also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes. An agent runs on each worker node and reports on the tasks assigned to it. The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

## **Manager nodes**

Manager nodes handle cluster management tasks:

* maintaining cluster state
* scheduling services
* serving swarm mode [HTTP API endpoints](https://docs.docker.com/engine/api/)

Managers nodes must have fixed ips.

When you have multiple managers you can recover from the failure of a manager node without downtime.

1. A three-manager swarm tolerates a maximum loss of one manager.
2. A five-manager swarm tolerates a maximum simultaneous loss of two manager nodes.
3. An N manager cluster will tolerate the loss of at most (N-1)/2 managers.
4. Docker recommends a maximum of seven manager nodes for a swarm.

## **Worker nodes**

Worker nodes are also instances of Docker Engine whose sole purpose is to execute containers. Worker nodes don’t participate in the Raft distributed state, make in scheduling decisions, or serve the swarm mode HTTP API.

## **Set up your swarm**

Manager: docker swarm init --advertise-addr <myvm1 ip>"

Workers: docker swarm join --token <token> <myvm ip>:<port>

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

**Take down the swarm.**

docker swarm leave --force

To avoid interference with manager node operation, you can drain manager nodes to make them unavailable as worker nodes:

docker node **update** *--availability drain <NODE>*

**When you drain a node, the scheduler reassigns any tasks running on the node to other available worker nodes in the swarm. It also prevents the scheduler from assigning tasks to the node.**

**Monitor swarm health**

You can monitor the health of manager nodes by querying the docker nodes API in JSON format through the /nodes HTTP endpoint.

From the command line, run docker node inspect <id-node> to query the nodes. For instance, to query the reachability of the node as a manager:

docker node inspect manager1 --format "{{ .ManagerStatus.Reachability }}"

reachable

To query the status of the node as a worker that accept tasks:

docker node inspect manager1 --format "{{ .Status.State }}"

ready

## Troubleshoot a manager node

You should never restart a manager node by copying the raft directory from another node. The data directory is unique to a node ID. A node can only use a node ID once to join the swarm. The node ID space should be globally unique.

To cleanly re-join a manager node to a cluster:

1. To demote the node to a worker, run docker node demote <NODE>.
2. To remove the node from the swarm, run docker node rm <NODE>.
3. Re-join the node to the swarm with a fresh state using docker swarm join.

## **Forcibly remove a node**

In most cases, you should shut down a node before removing it from a swarm with the docker node rm command. If a node becomes unreachable, unresponsive, or compromised you can forcefully remove the node without shutting it down by passing the –force flag. For instance, if node9 becomes compromised:

## **Back up the swarm**

Docker manager nodes store the swarm state and manager logs in the /var/lib/docker/swarm/ directory. In 1.13 and higher, this data includes the keys used to encrypt the Raft logs. Without these keys, you will not be able to restore the swarm.

Service: <https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/>

When you create a service, you specify which container image to use and which commands to execute inside running containers. You also define options for the service including:

* the port where the swarm will make the service available outside the swarm
* an overlay network for the service to connect to other services in the swarm
* CPU and memory limits and reservations
* a rolling update policy
* **the number of replicas of the image to run in the swarm**